Introduction to Message Passing Interface (MPI)

Bruno C. Mundim

SciNet HPC Consortium

December 4, 2024

About this course

- We'll review some basic concepts of "supercomputing", a.k.a. high performance computing (HPC) and Message Passing Interface (MPI)
- It is intended to be a high level primer for those largely new to HPC and MPI.
- Topics will include motivation for HPC and distributed memory computing, problem characteristics as they apply to parallelism and what kind of problems MPI addresses.
- Some familiarity with the Linux command line, editing text files, and C/C++ or FORTRAN is preferred.

What do you need for the course?

- A computer with browser and internet connection to attend the lectures.
- A Zoom client to connect to the virtual lectures.
- An ssh client to connect to the SciNet Teach cluster.
  - Linux and MaxOS: Use the ssh command in the terminal.
  - Windows: Use MobaXTerm https://mobaxterm.mobatek.net.

Make sure you can login to the website https://scinet.courses/1368/ !

Course structure

- MONDAY: A first online lecture over Zoom.

  My colleague Alexey Fedoseev introduced OpenMP and shared memory systems. An assignment was given at the end of the lecture.

- WEDNESDAY: A second online lecture over Zoom.

  I will introduce MPI and distributed memory systems. An assignment will also be given at the end of the lecture.

  Submit a solution for the assignment on the course website (deadline is midnight Thursday).

  Reminder: you can ask questions:

  - in the Zoom chat during and at the end of the lecture.
  - in the student forum on the course site.

- FRIDAY: A last online lecture on Zoom that will address both solutions, common mistakes, and wrap-up.

Course Outline

- Distributed Memory Computing
- MPI: Basics
- MPI: Send & Receive
- MPI: Collectives

Distributed Memory Computing

HPC Systems

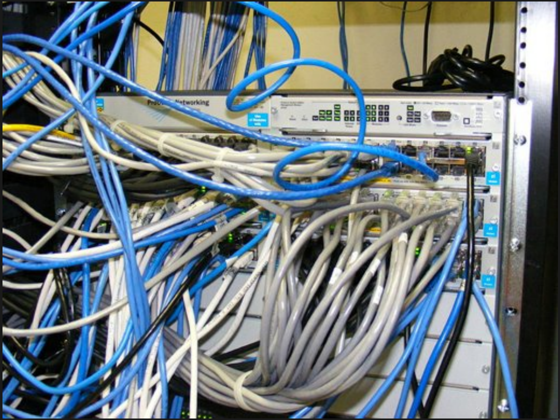
Architectures

- Vector machines
  - No longer dominant in HPC anymore.
  - Cray, NEC
- Symmetric Multiprocessor (SMP) machines, or, shared memory machines
  - Processors can all see and use the same memory. Typically a limited number of cores.
  - Present in virtually all systems these days.
- Accelerator devices (GPGPU, Cell, MIC, FPGA)
  - Heterogeneous use of standard CPU's with a specialized off-host accelerator.
  - NVIDIA, AMD (Xilinx), Intel (Altera).
- Clusters, or, distributed memory machines
  - A bunch of servers linked together by a network ("interconnect").
  - GigE, Infiniband, Cray Gemini/Aries, IBM BGQ Torus
- Hybrid machines (Modern HPC clusters)
  - Hybrid combo of these different architectures.

Distributed Memory: Clusters

Simplest type of parallel computer to build

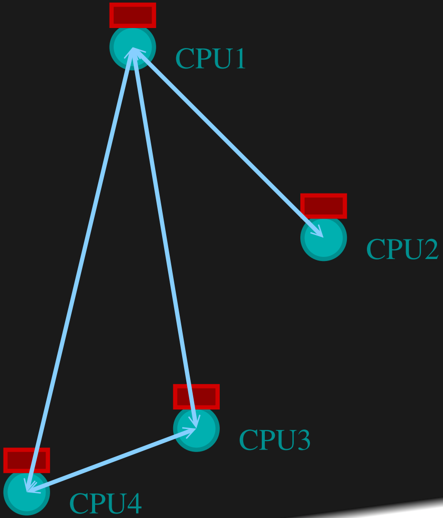
- Take existing powerful standalone computers
- And network them



(source: http://flickr.com/photos/eurleif)

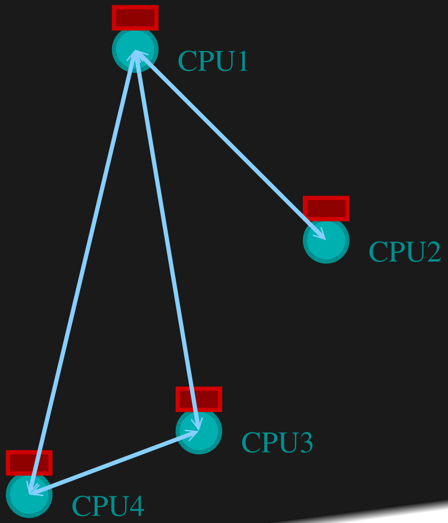
Distributed Memory: Clusters

- Each node is independent!
  - Parallel code consists of programs running on separate computers, communicating with each other.
  - Could be entirely different programs.
- Each node has its own memory!
  - Whenever it needs data from another region, requests it from that CPU.
  - Usual model: "message passing"



Clusters+Message Passing

- Hardware:
  - Easy to build (Harder to build well)
  - Can build larger and larger clusters relatively easily
- Software:
  - Every communication has to be hand-coded: hard to program



HPC Programming Models

Parallel Programming Approaches

- Serial (embarrassingly parallel applications)
  - C, C++, Fortran, Julia, Bash or Python Scripting Languages
- Threads (shared memory systems)
  - OpenMP, pthreads
- Heterogeneous computing (off-host accelerators: GPGPU, Cell, MIC, FPGA)
  - CUDA, OpenCL, OpenACC, and OpenMP
- Message passing (distributed memory systems)
  - MPI, PGAS (UPC, Coarray Fortran)
- Hybrid combinations of the above

We will focus on MPI programming in this lecture.

MPI: Basics

Message Passing Interface (MPI)

What is it?

- An open standard library interface specification for message passing, ratified by the MPI Forum
- Version: 1.0 (1994), 1.1 (1995), 1.2 (1997), 1.3 (2008)
- Version: 2.0 (1997), 2.1 (2008), 2.2 (2009)
- Version: 3.0 (2012), 3.1 (2015)
- Version: 4.0 (2021)

MPI Implementations

- OpenMPI www.open-mpi.org
  - Alliance clusters (Graham, Cedar, Niagara, etc...):
    `module load gcc openmpi`
    or
    `module load intel openmpi`

  Currently these give you OpenMPI version 4.0.3.
- MPICH2 www.mpich.org
  - MPICH 3.x, MVAPICH2 2.x , IntelMPI 2019.x
  - Alliance clusters: module load intel intelmpi

MPI is a Library for Message Passing

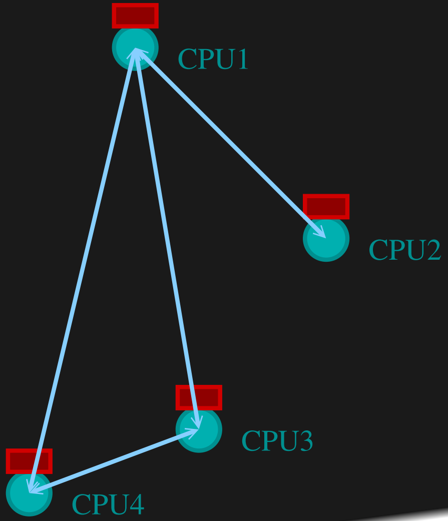
- Not built into the compiler.
- Function calls that can be made from any compiler, many languages.
- Just link to it.
- Wrappers: mpicc, mpif90, mpicxx

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, err;
  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from task %d of %d!\n",rank,
                                           size);
  err = MPI_Finalize();
}
```

```fortran
program helloworld
use mpi
implicit none
integer :: rank, commsize, err
call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, commsize, err)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
print *,'Hello world from task',rank,'of',commsize
call MPI_Finalize(err)
end program helloworld
```

MPI is a Library for Message Passing

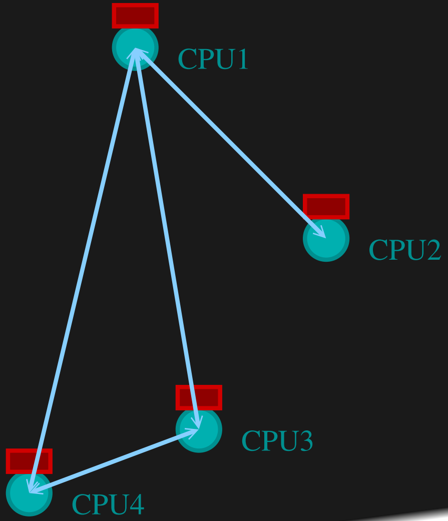
- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.



MPI is a Library for Message Passing

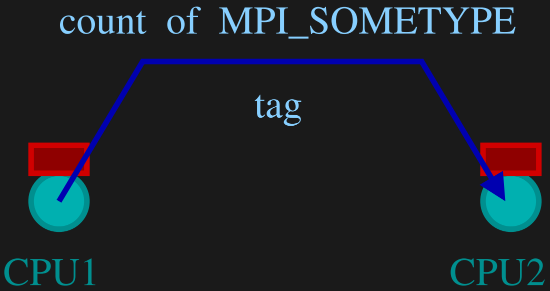
Three basic sets of functionality:

- Pairwise communications via messages
- Collective operations via messages
- Efficient routines for getting data from memory into messages and vice versa



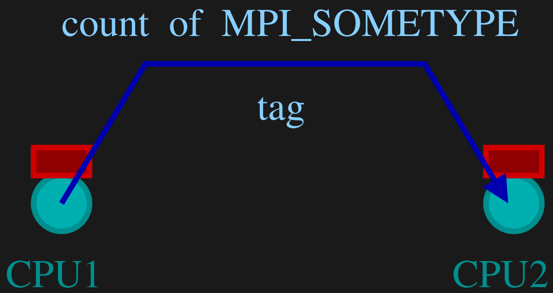
Messages

- MPI messages are a string of length **count** all of some fixed MPI **type**.
- MPI types exist for characters, integers, floating point numbers, etc.
- An arbitrary non-negative integer **tag** is also included – it helps keep things straight if lots of messages are sent.



Messages

- Messages have a **sender** and a **receiver**.
- When you are sending a message, don't need to specify sender (it's the current processor).
- A sent message has to be actively received by the receiving process.



Size of MPI Library

- Many, many functions (>200)
- Not nearly so many concepts
- We'll get started with just 10-12, use more as needed.

```
MPI_Init()

MPI_Comm_size()

MPI_Comm_rank()

MPI_Ssend()

MPI_Recv()

MPI_Finalize()
```

Access to SciNet's Teach supercomputer

Access to SciNet's Teach supercomputer

- SciNet's Teach supercomputer is part of the old GPC system (42 nodes) that has been repurposed for education and training in general, and in particular for many of summer school sessions.
- Log into Teach login node, **teach01**, with your lcl_uothpc163sNNNN temporary account.

```
$ ssh -Y lcl_uothpc163sNNNN@teach.scinet.utoronto.ca
$ cd $SCRATCH
$ cp -r /scinet/course/mpi/2024/hpc163 .
$ cd hpc163
$ source setup
```

Running computations

- On most supercomputers, a scheduler governs the allocation of resources.
- This means submitting a job with a jobscript.
- `srun`: a command that is a resource request + job running command all in one, and will run the command on one (or more) of the available resources.
- We have set aside 34 nodes with 16 cores for this class, so occasionally, only in very busy sessions, you may have to wait for someone else's `srun` command to finish.

Example: Hello World

- The obligatory starting point
- cd hpc163/mpi-intro
- Compile and run it together

C:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from task %d of %d!\n",rank,
                                              size);
  MPI_Finalize();
}
```

Fortran:

```
program helloworld
use mpi
implicit none
integer :: rank, commsize, err
call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, commsize, err)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
print *,'Hello world from task',rank,'of',commsize
call MPI_Finalize(err)
end program helloworld
```

```
$ source $SCRATCH/hpc163/setup
$ mpif90 hello-world.f90 -o hello-worldf
or
$ mpicc hello-world.c -o hello-worldc
$ srun -n 1 hello-worldc
$ srun -n 2 hello-worldc
$ srun -n 8 hello-worldc
```

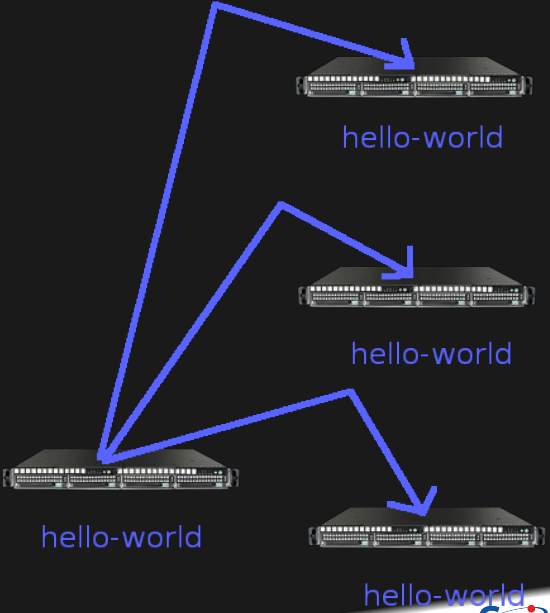
What does mpicc/mpif90 do?

- Just wrappers for the regular C, Fortran compilers that have the various -I, -L clauses in there automaticaly.
- `--showme` (OpenMPI) shows which options are being used.

```
$ mpicc --showme hello-world.c -o hello-worldc
gcc hello-world.c -o hello-world -I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/ope
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/openmpi/opal/mca/hwloc/hwloc1117/hwl
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/openmpi/opal/mca/event/libevent2022/
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/openmpi/opal/mca/event/libevent2022/
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include -pthread -L/opt/slurm/lib64
-L/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/hcoll/lib -L/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/m
-L/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/ucx/lib -Wl,-rpath -Wl,/opt/slurm/lib64 -Wl,-rpath
-Wl,/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/hcoll/lib -Wl,-rpath
-Wl,/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/mxm/lib -Wl,-rpath
-Wl,/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/ucx/lib -Wl,-rpath
-Wl,/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/lib -Wl,--enable-new-dtags
-L/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/lib -lmpi
$
```

What mpirun/srun does

- Launches *n* processes, assigns each an MPI **rank** and starts the program
- Usually, the processes run the same executable, therefore **each process runs the exact same code**.
- For multinode run, has a list of nodes, ssh's to each node and launches the program
- `mpirun` only runs the processes on the login node, and does not allocate resources; typically used inside a batch job.
- `srun` allocates the resources on the cluster and runs the processes there: This is what we'll use in this class.



Number of Processes

- Number of processes to use is almost always equal to the number of processor's cores on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not available on Teach cluster).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

In this session, omit the `-N` argument and use `srun` with a `-n` argument only.

Regular pure mpi run on a 40-core node:

```
$ srun -N 1 -n 40 hello-worldc
```

Hyperthreaded mpi run (not on Teach):

```
$ srun -N 1 -n 80 hello-worldc
```

Memory-hungry mpi run on a 40-core node requiring 8GB per process:

```
$ srun -N 1 -n 20 hello-worldc
```

Hybrid run (8 mpi processes with 5 threads):

```
$ srun -N 1 -n 8 -c 5 hello-worldc
```

mpirun / srun runs any program

- mpirun will start that process launching procedure for any program
- Sets variables somehow that mpi programs recognize so that they know which process they are.

```
$ hostname
teach01
$ mpirun -n 2 hostname
teach01
teach01
$ srun -n 2 hostname
teach39
teach39
$
```

Example: “Hello World”

```
$ srun -n 4 ./hello-worldc
Hello from task 2 of 4 world
Hello from task 1 of 4 world
Hello from task 0 of 4 world
Hello from task 3 of 4 world
```

```
$ srun --label -n 4 ./hello-worldc
2: Hello from task 2 of 4 world
1: Hello from task 1 of 4 world
0: Hello from task 0 of 4 world
3: Hello from task 3 of 4 world
```

```
$ mpirun --tag-output -n 4 ./hello-worldc
[1,2]<stdout>:Hello from task 2 of 4
[1,3]<stdout>:Hello from task 3 of 4
[1,0]<stdout>:Hello from task 0 of 4
[1,1]<stdout>:Hello from task 1 of 4
```

The `--tag-output` flag is specific for the OpenMPI implementation of MPI.

Make

- Make builds an executable from a list of source code files and rules
- Many files to do, of which order doesn't matter for most
- Parallelism!
- `make -j N` launches N processes to do it.

```
$ make
$ make -j 2
$ make -j
```

What the code does (Fortran)

```fortran
program helloworld
use mpi
implicit none
integer :: rank, commsize, err

call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, commsize, err)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)

print *,'Hello world from task',rank,'of',commsize

call MPI_Finalize(err)
end program helloworld
```

- `use mpi`: imports declarations for MPI function calls
- `call MPI_INIT(err)`: initialization for MPI library. Must come first.
- `err`: Returns any error code.
- `call MPI_FINALIZE(err)`: close up MPI stuff. Must come last. err: Returns any error code.
- `call MPI_COMM_RANK`, `call MPI_COMM_SIZE`: requires a little more exposition.

What the code does (C)

- `#include <mpi.h>` - MPI library definitions
- `MPI_Init(&argc,&argv)`
  MPI Intialization, must come first
- `MPI_Finalize()`
  Finalizes MPI, must come last
- `err` - MPI routine could return an error code. In practice, MPI applications abort when there is an error.

Communicator Components

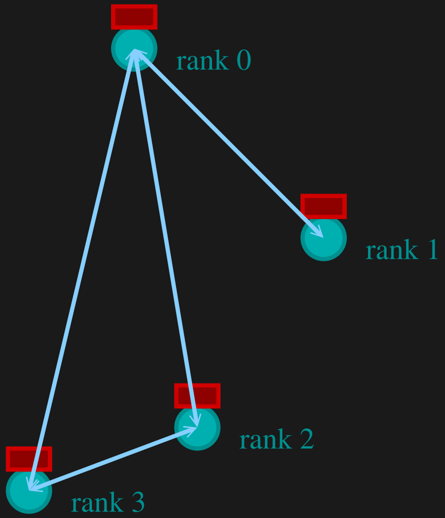
- A communicator is a handle to a group of processes that can communicate.
- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
- `MPI_Comm_size(MPI_COMM_WORLD,&size)`

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {

  int rank, size;
  int err;

  err = MPI_Init(&argc, &argv);

  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello world from task %d of %d!\n",rank,
                                             size);

  MPI_Finalize();

}
```

Communicators

- MPI groups processes into communicators.
- Each communicator has some size – number of tasks.
- Every task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`.

`MPI_COMM_WORLD`:
size = 4, ranks = 0..3

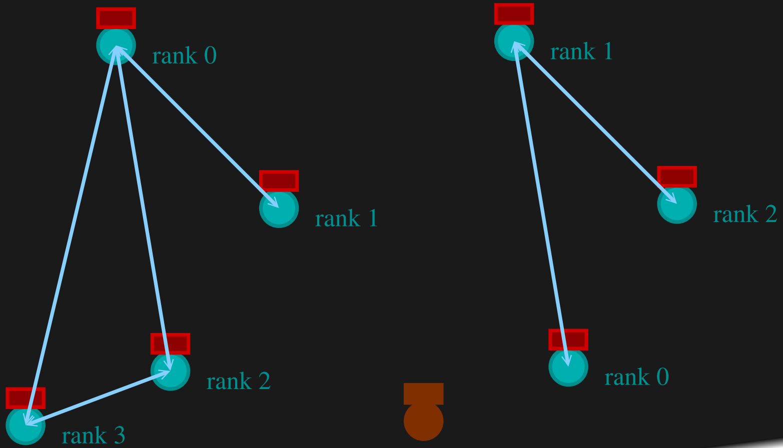


Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason

`MPI_COMM_WORLD`:
size=4,ranks=0..3

`new_comm`:
size=3,ranks=0..2



MPI Communicator Basics

Communicator Components

- `MPI_COMM_WORLD`:
  Global Communicator
- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
  Get current task's rank
- `MPI_Comm_size(MPI_COMM_WORLD,&size)`
  Get communicator size

Send & Receive

MPI: Send & Receive

`hello-world` was our first real MPI program
But no Messages were being Passed.

- Let's fix this
- `mpicc -o firstmessagec firstmessage.c`
- `srun -n 2 ./firstmessagec`
- Note C: MPI_CHAR

```c
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  int sendto, recvfrom;  /*task to send,recv from*/
  int ourtag=1;          /*tag to label msgs*/
  char sendmsg[]="Hello";/*text to send*/
  char getmsg[6];        /*text to receive*/
  MPI_Status rstatus;    /*recv status info*/
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    sendto = 1;
    MPI_Ssend(sendmsg, 6, MPI_CHAR, sendto,
              ourtag, MPI_COMM_WORLD);
    printf("%d: Sent msg <%s>\n",rank,sendmsg);
  } else if (rank == 1) {
    recvfrom = 0;
    MPI_Recv(getmsg, 6, MPI_CHAR, recvfrom,
             ourtag, MPI_COMM_WORLD, &rstatus);
    printf("%d: Got msg <%s>\n", rank, getmsg);
  }
  MPI_Finalize();
}
```

MPI: Send & Receive

- Let's fix this, Fortran version
- `mpif90 -o firstmessagef firstmessage.f90`
- `srun -n 2 ./firstmessagef`
- Note Fortran: MPI_CHARACTER

```fortran
program firstmessage
use mpi
implicit none
integer :: rank,comsize,err
integer :: sendto,recvfrom  !Task to send,recv from
integer :: ourtag=1         !tag to label msgs
character(5) :: sendmessage !text to send
character(5) :: getmessage  !text rcvd
integer, dimension(MPI_STATUS_SIZE) :: rstatus
call MPI_Init(err)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, err)
if (rank == 0) then
 sendmessage = 'Hello'
 sendto = 1
 call MPI_Ssend(sendmessage,5,MPI_CHARACTER,sendto,&
               ourtag,MPI_COMM_WORLD,err)
 print *, rank, ' sent message <',sendmessage,'>'
else if (rank == 1) then
 recvfrom = 0
 call MPI_Recv(getmessage,5,MPI_CHARACTER,recvfrom,&
              ourtag,MPI_COMM_WORLD,rstatus,err)
 print *, rank, ' got message <',getmessage,'>'
endif
call MPI_Finalize(err)
end program firstmessage
```

Send and Receive

C

```
MPI_Status status;
err = MPI_Ssend(sendptr, count, MPI_TYPE, destination, tag, Communicator);
err = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag, Communicator, status);
```

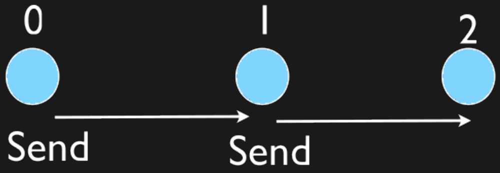
Note: MPI has a Fortran and C interface. We can use the C interface in C++ but will have to deal with pointers, i.e., we'll give arguments likes `&(array[0])` or `array.data()` instead of just `array`.

Fortran

```
integer status(MPI_STATUS_SIZE)
call MPI_SSEND(sendarr, count, MPI_TYPE, destination, tag, Communicator, err)
call MPI_RECV(rcvarr, count, MPI_TYPE, source, tag, Communicator, status, err)
```

MPI Communication Patterns

Send a message to the right:



Specials

Special Source/Destination `MPI_PROC_NULL`

`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

Special Source `MPI_ANY_SOURCE`

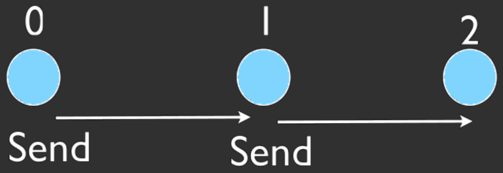
`MPI_ANY_SOURCE` is a wildcard; matches any source when receiving.

Special Status `MPI_STATUS_IGNORE`

Use `MPI_STATUS_IGNORE` if you do not want to capture the status in a receive.

MPI: Send Right, Receive Left

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv) {
    int          rank, size, err, left, right, tag = 1;
    double       msgsent, msgrcvd;
    MPI_Status   rstatus;
    err = MPI_Init(&argc, &argv);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    err = MPI_Comm_size(MPI_COMM_WORLD, &size);
    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;
    right = rank + 1;
    if (right >= size) right = MPI_PROC_NULL;
    msgsent = rank*rank;
    msgrcvd = -999.;
    err = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    cout << to_string(rank) + ": Sent " + to_string(msgsent) + " and got " + to_string(msgrcvd) + "\n";
    err = MPI_Finalize();
}
```



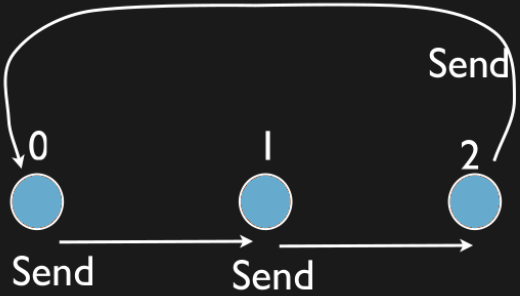
MPI: Send Right, Receive Left

```
$ make secondmessagec
$ srun -n 3 ./secondmessagec
2: Sent 4.000000 and got 1.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
$
```

```
$ srun -n 6 ./secondmessagec
4: Sent 16.000000 and got 9.000000
5: Sent 25.000000 and got 16.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
```

MPI: Send Right, Receive Left with Periodic BCs

Periodic Boundary Conditions:



MPI: Send Right, Receive Left with Periodic BCs

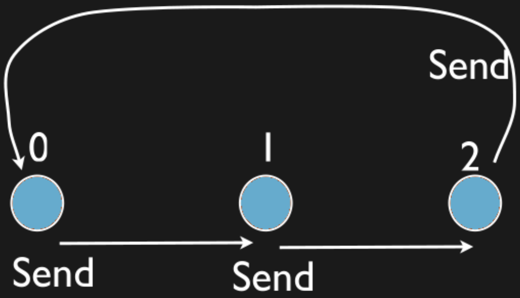
```
    ...
    left = rank - 1;
    if (left < 0) left = size-1; // Periodic BC
    right = rank + 1;
    if (right >= size) right =0; // Periodic BC
    msgsent = rank*rank;
    msgrcvd = -999.;
    ...
```

```
$ make thirdmessagec # or thirdmessagef
$ srun -n 5 thirdmessagec
```

Just sort of hangs there doing nothing?

Deadlock!

- A classic parallel bug.
- Occurs when a cycle of tasks are waiting for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) a deadlock.
- Here, all processes are waiting for the send to complete, but no one is receiving.



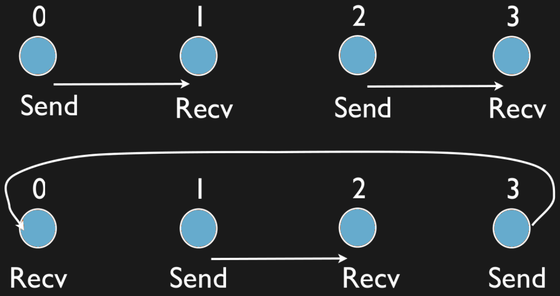
Big MPI Lesson #1

All sends and receives must be paired at the time of sending

How do we fix the deadlock?

Without using new MPI routine, how do we fix the deadlock?

Even-odd solution



- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd number of processes? How about 2? 1?

MPI: Send Right, Receive Left with Periodic BCs - fixed

```
    ...
    if ((rank % 2) == 0) {
        err =  MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
        err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    } else {
        err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
        err =  MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    }
    ...
```

```
$ make fourthmessagec
$ srun -n 5 ./fourthmessagec
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

MPI: Sendrecv

```
err = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,
                   recvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Status)
```

- A blocking send and receive built together
- Let them happen simultaneously
- Can automatically pair send/recvs
- Why 2 sets of tags/types/counts?

Send Right, Receive Left with Periodic BCs - Sendrecv

Code

```
    ...
    err =  MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE, right, tag,
              &msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    ...
```

Execution

```
$ make fifthmessagec
$ srun -n 5 ./fifthmessagec
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

Different versions of SEND

MPI_Ssend: Standard synchronous send

- guaranteed to be synchronous.
- routine will not return until the receiver has "picked up' '.

MPI_Bsend: Buffered Send

- guaranteed to be asynchronous.
- routine returns before the message is delivered.
- system copies data into a buffer and sends it in due course.
- can fail if buffer is full.

MPI_Send (standard Send)

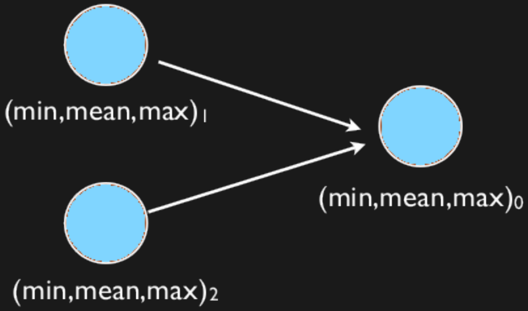
- may be implemented as synchronous or asynchronous send.
- causes a lot of confusion.

In this class, stick with MPI_Ssend for clarity and robustness

Collectives

Reductions: Min, Mean, Max Example

- Calculate the min/mean/max of random numbers -1.0 . . . 1.0
- Should trend to -1/0/+1 for a large N.
- How to MPI it?
- Partial results on each node, collect all to node 0.



Reductions: Min, Mean, Max Example

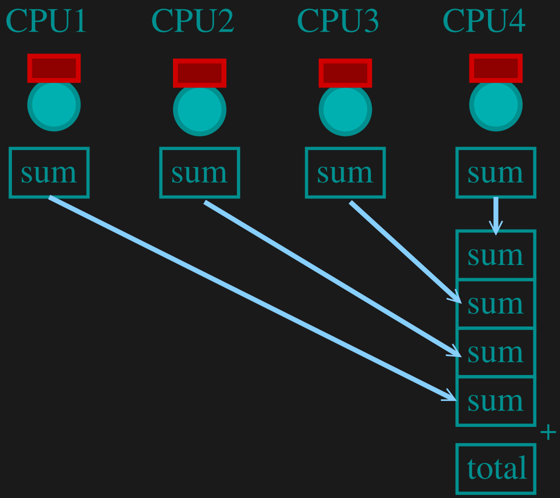
```
#include <mpi.h>
#include <iostream>
#include <algorithm>
#include <cstdlib>
using namespace std;
int main(int argc, char **argv) {
   const int nx = 1500, MIN=0, MEAN=1, MAX=2;
   double mmm[3] = {1e+19, 0, -1e+19};
   int rank, size, tag = 1;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   double *dat = new double[nx];
   srand(0);
   for (int i=0;i<dx*rank;i++) rand();
   for (int i=0;i<nx;i++)
      dat[i] = 2*((double)rand()/RAND_MAX)-1.;
   for (int i=0;i<nx;i++) {
      mmm[MIN] = min(dat[i], mmm[MIN]);
      mmm[MAX] = max(dat[i], mmm[MAX]);
      mmm[MEAN] += dat[i];
   }
   mmm[MEAN] /= nx;
```

```
   if (rank != 0)
      MPI_Ssend(mmm, 3, MPI_DOUBLE, 0, tag,
                MPI_COMM_WORLD);
   else {
      double recvmmm[3];
      for (int i=1;i<size;i++) {
         MPI_Recv(recvmmm, 3, MPI_DOUBLE,
                  MPI_ANY_SOURCE, tag,
                  MPI_COMM_WORLD,MPI_STATUS_IGNORE);
         mmm[MIN] = min(recvmmm[MIN], mmm[MIN]);
         mmm[MAX] = max(recvmmm[MAX], mmm[MAX]);
         mmm[MEAN] += recvmmm[MEAN];
      }
      mmm[MEAN] /= size;
      cout << "Global Min/mean/max " << mmm[MIN] <<
              globmmm[MEAN]<<" "<<mmm[MAX]<<endl;
   }
   MPI_Finalize();
}
```

Inefficient!

- Requires (P-1) messages
- 2(P-1) if everyone then needs to get the answer.

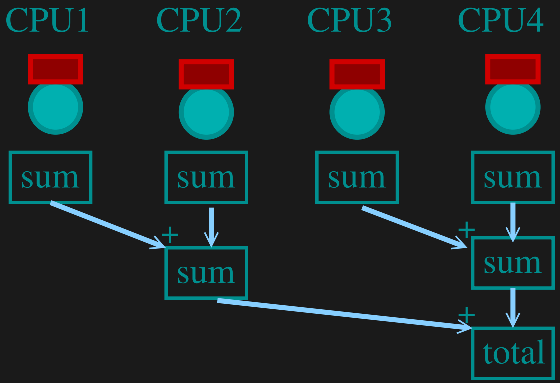
$$T_{comm} = PC_{comm}$$



Better Summing

- Pairs of processors; send partial sums
- Max messages received $\log_2(P)$
- Can repeat to send total back.

$$T_{comm} = 2\log_2(P)C_{comm}$$



Reduction: Works for a variety of operations (+,*,min,max)

MPI Collectives

```
err = MPI_Allreduce(sendptr, rcvptr, count, MPI_TYPE, MPI_Op, Communicator);
err = MPI_Reduce(sendbuf, recvbuf, count, MPI_TYPE, MPI_Op, root, Communicator);
```

- `sendptr/rcvptr`: pointers to buffers
- `count`: number of elements in ptrs
- `MPI_TYPE`: one of `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, `MPI_CHAR`, etc.
- `MPI_Op`: one of `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`.
- `Communicator`: MPI_COMM_WORLD or user created.
- `All` variants send result back to all processes; non-`All` sends to process `root`.

Reductions: Min, Mean, Max with MPI Collectives

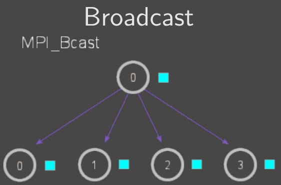
```
double globalmmm[3];
MPI_Allreduce(&mmm[MIN], &globalmmm[MIN], 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
MPI_Allreduce(&mmm[MAX], &globalmmm[MAX], 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&mmm[MEAN], &globalmmm[MEAN], 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
globalmmm[MEAN] /= size;
if (rank==0)
   cout << "Global Min/mean/max " << mmm[MIN] << " " <<
           globmmm[MEAN]<<" "<<mmm[MAX] << endl;
```

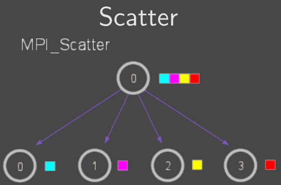
Collective Operations

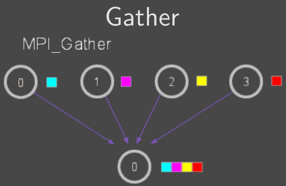
Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

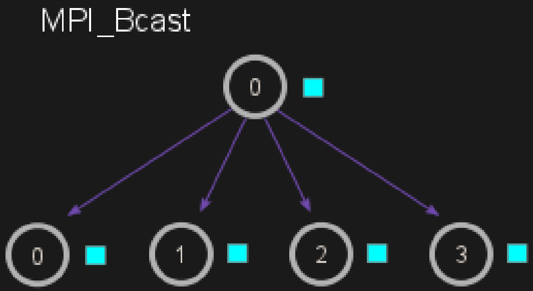






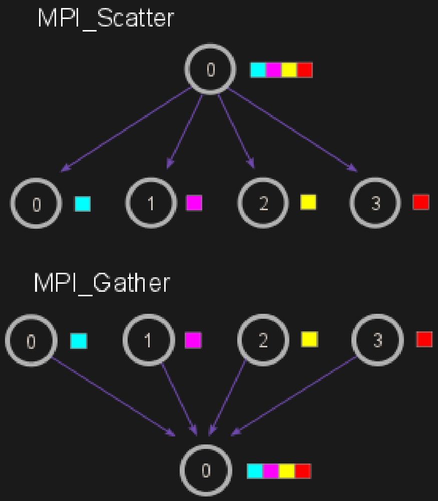
- File I/O
- Barriers (don't!)
- All-to-all ...

MPI_Collectives: Broadcast



- Broadcasts a message from process with rank “root” to all processes in group, including itself.
- Amount of data sent must be equal to amount of data received.
- `err = MPI_Bcast(void *buf, count, MPI_Type, root, Comm)`
  - `buf`: buffer of data to send/recv
  - `count`: number of elements in buf
  - `MPI_TYPE`: one of `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, `MPI_CHAR`, etc.
  - `root`: “root” processor to send from
  - Communicator: MPI_COMM WORLD or user created

MPI_Collectives: Scatter/Gather



- Scatter: Sends data from "root" to all processes in group.
- `err = MPI_Scatter(void *send_buf, send_count, MPI_Type, void *recv_buf, recv_count, MPI_Type, root, Comm)`
- Gather: Receives data on "root" from all processes in group.
- `err = MPI_Gather(void *send_buf, send_count, MPI_Type, void *recv_buf, recv_count, MPI_Type, root, Comm)`

Example: Scatter/Gather

Scatter

Simple Scatter example sending data from root to 4 procesors.

```
$ cd $SCRATCH/hpc163/collectives
$ make
$ srun -n 4 ./scatter
```

Gather

- Copy Scatter.c to Gather.c and reverse the process.
- Send from 4 processes and collect on root using `MPI_Gather()`.

MPI_Collectives: Barrier

- Blocks calling process until all group members have called it.
- Decreases performance. Try to avoid using it explicitly.
- `err = MPI_Barrier(Comm)`
    - Communicator Comm: `MPI_COMM WORLD` or user created

MPI_Collectives: All-to-all

```
int MPI_Alltoall(const void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf, int recvcount,
            MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
            RECVTYPE, COMM, IERROR)

            <type>     SENDBUF(*), RECVBUF(*)
            INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE
            INTEGER    COMM, IERROR
```

- `MPI_Alltoall` is a collective operation in which all processes send the same amount of data to each other, and receive the same amount of data from each other.
- Each process breaks up its local sendbuf into n blocks (like `Scatter`), each containing sendcount elements of type sendtype, and divides its recvbuf similarly according to recvcount and recvtype (like `Gather`).

MPI Summary

MPI Summary - C syntax

```
MPI_Status status;

err = MPI_Init(&argc, &argv);

err = MPI_Comm_{size,rank}(Communicator, &{size,rank});

err = MPI_Send(sendptr, count, MPI_TYPE, destination, tag, Communicator);

err = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag, Communicator, &status);

err = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination,tag, recvptr, count, MPI_TYPE, source,
                   tag, Communicator, &status);

err = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE, MPI_OP, Communicator);

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

MPI Summary - FORTRAN syntax

```
integer status(MPI_STATUS_SIZE)

call MPI_INIT(err)

call MPI_COMM_{SIZE,RANK}(Communicator, {size,rank},err)

call MPI_SSEND(sendarr, count, MPI_TYPE, destination, tag, Communicator)

call MPI_RECV(rcvarr, count, MPI_TYPE, destination,tag, Communicator, status, err)

call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination,tag, recvptr, count, MPI_TYPE, source, &
                  tag, Communicator, status, err)

call MPI_ALLREDUCE(mydata, globaldata, count, MPI_TYPE, MPI_OP, Communicator, err)

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_REAL, MPI_DOUBLE_PRECISION, MPI_INTEGER, MPI_CHARACTER
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

Conclusion

Recap

- Distributed Memory Computing
- MPI: Basics
- MPI: Send & Receive
- MPI: Collectives

Good References

- W. Gropp, E. Lusk, and A. Skjellun, Using MPI: Portable Parallel Programming with the Message-Passing Interface. Third Edition. (MIT Press, 2014).
- W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface. (MIT Press, 2014).
- The man pages for various MPI commands.
- http://www.mpi-forum.org/docs/